The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04872-y

A cluster-based practical key recovery attack on reduced-round AES using impossible-differential cryptanalysis

Debranjan Pal[1] · Md Rasid Ali[1] · Abhijit Das[1] · Dipanwita Roy Chowdhury[1]

Accepted: 5 October 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Advanced Encryption Standard (AES) is a well-known, extensively used, broadly studied, and industry-standard symmetric key-based block cipher. Hence cryptanalysis of AES is a felicitous job for the cryptography researchers. Impossible differential cryptanalysis (IDC) is a powerful attack technique on symmetric-key ciphers. In this paper, we report a cluster-based parallel attack to retrieve the full key for all versions of five-round AES (128, 192, 256) and partial key recovery of six-round AES (192, 256) using IDC. The full key of AES-128 can be determined in only 6.5 min. To recover the full key of five-round AES-192/256, a total of 10 minutes is required. Next we find the second round key of six-round AES-(192, 256). It takes a total of 12.5 min. To the best of our knowledge, our results outperformed the existing best results with respect to timing and as well as the success probability for the full/partial key recovery of five/six rounds of AES.

Keywords AES · Chosen plaintext attack · Block cipher · Impossible-differential cryptanalysis · Full-key recovery · Parallel attack

Md Rasid Ali, Abhijit Das and Dipanwita Roy Chowdhury have contributed equally to this work.

✉ Debranjan Pal
debranjanpal@iitkgp.ac.in

Md Rasid Ali
rasid@iitkgp.ac.in

Abhijit Das
abhij@cse.iitkgp.ac.in

Dipanwita Roy Chowdhury
drc@cse.iitkgp.ac.in

[1] Crypto Research Lab, Department of Computer Science and Engineering, IIT Kharagpur, Kharagpur, West Bengal 721301, India

Published online: 02 November 2022

1 Introduction

Symmetric key ciphers are cryptographic algorithms that apply same key for both encryption and decryption processes. Symmetric ciphers are two types, block and stream. The designers construct block ciphers by repeating a special function multiple times, called round function. Cryptanalysis is a technique to assess the strength of cryptographic components and protocols. Differential cryptanalysis [1] is the first general technique for the analysis of iterated block ciphers. In 1990, Biham et al. propose the concept of differential cryptanalysis. It is a chosen-plaintext attack, where the attacker can choose some input plaintexts and try to examine the output ciphertexts in an attempt to retrieve the key. We define the differential characteristic as a sequence of input and output differences such that the output difference of one round comes from the input differences from the previous round. A highly likely differential characteristic in a cipher path provides a scope to calculate the last-round partial keys by processing the information from the last round. Impossible differential cryptanalysis (IDC) is a special type of differential cryptanalysis for block ciphers. IDC tries to impose nonzero differentials in the cipher path that occur with probability zero. The round keys for which such incidents occur can be eliminated, thus reducing the keyspace. A brute-force search can be applied in the reduced keyspace to retrieve the actual round key.

Knudsen proposed impossible differential cryptanalysis [2] in 1998. In Crypto 1998, Biham et al. [3] first mentioned the name impossible differential and applied this to mount attack on 4.5 rounds of IDEA and 31 rounds of Skipjack. Recently IDC has been proved as one of the dominant form of attack against the block ciphers. In [4], the attack is employed along with multiple differential paths, and applied on LBlock, CLEFIA-128, Simon and Camellia. The relations between data and time complexity of impossible differential attack are presented in [5, 6]. The most well-adapted and successful attack on AES is the impossible differential cryptanalysis (IDC) [3]. Surprisingly, most of the impossible differential attacks since [7] cannot be attacked practically due to high data and memory complexities. In this paper, we focus practical attack of AES-128, AES-192, and AES-256 using IDC. We execute the whole attack process in a cluster. A cluster is a collaboration of single systems/processors, which are linked together through high-speed LAN's and the whole platform works as a single coherent system.

1.1 Revisiting the attacks on AES

Here we distinguish among the different attacks on AES.

1.1.1 Collision attack

Gilbert et al. [8] introduced this attack on AES. They found an interesting property between four inner rounds of Rijndael and a random permutation of the block space, by utilizing the presence of collisions between some partial functions persuaded

by the cipher. The four rounds distinguisher allows them to attack AES reduced to seven rounds which require a data complexity of 2^{32} and the time complexity is 2^{140}. The three rounds distinguisher is restricted up to six rounds with all available key sizes of Rijndael. Though this attack does not affect vividly on the security of the cipher but indicate any flaw in the design and prove any inadequacy in the selection of the number of rounds.

1.1.2 Square attack

Consider a set of 256 plaintext states, where all bytes have same value (called passive byte) except one byte (called active byte), which consists of all the 2^8 possible values. Such a set is called δ-set. A byte position in a δ-set either contains same value or contains all the 256 values. Hence XOR-sum of all the 256 values corresponding to any byte results zero. This property is called balancedness property. This property can directly be applied to four-rounds of AES. Applying a δ-set as input to a four-round AES, up to three rounds it maintains balancedness property. This property vanishes after the application of sub-byte transformation of fourth round. This property can be applied to attack four round AES. Guess any one byte of fourth round key and use the corresponding ciphertext byte to decrypt one round for a δ-set. If the XOR sum value is zero, then the fourth round key byte is correctly guessed. One can use this property to attack higher rounds of AES. The attack is called square attack as it is first applied to Square cipher [9].

1.1.3 Boomerang attack

1n 1999, Wagner [10] proposes boomerang attack. Let E:$\{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$ be an iterative block cipher that can be subdivided into two parts E_0, E_1 such that $E = E_1 \circ E_0$. Assume E_0 occurs with the differential trail a and E_1 occurs with differential trail b such that $a^2b^2 >> 2^{-n}$. Then one can distinguish the cipher E from a random permutation. Biryukov et al. [11] first use a boomerang distinguisher to mount related key attacks on AES-192 and AES-256. The recent boomerang attacks on different versions of AES is mentioned in Table 1.

1.1.4 Mixture differential cryptanalysis

Grassi [12] first introduced the concept of mixture differential cryptanalysis for attacking reduced rounds of AES. In this attack, choose a plaintext pair and generate a new set of pairs by applying the generating variables of the original plaintext pair. Grassi [12] proved that for four round AES, the ciphertexts generated from the original pair occur in a fixed subspace iff the ciphertext pairs for the newly generated plaintext pairs find the same property. The distinguisher is independent of key schedule, substitution, and mixcolumn matrix (except with branch number five). They use the distinguisher to attack reduced rounds of AES.

Table 1 Full Key recovery attacks on reduced round AES (ACP means adaptive chosen plaintexts)

Attack	AES version	Rounds	Data complexity	Time complexity	Memory	Success Prob.*
MixInt [15]	AES-128	3	2^{2}	$2^{8.1}$	–	0.65
MixInt [15]	AES-128	4	6	$2^{45.3}$	–	–
Square [21]	AES-128	4	2^{9}	2^{14}	Small	1
Boomeyong [22]	AES-128	5	$2^{49}ACC$	$2^{48}XOR$	2^{23}	–
Boomerang [11]	AES-128	5	2^{39}	2^{39}	2^{33}	–
IDC [7]	AES-128	5	$2^{29.5}$	2^{31}	2^{42}	–
IDC (Sect. 3)	AES-128	5	2^{34}	2^{31}	$2^{36.6}$	0.99
IDC (Sect. 4)	AES-192/256	5	2^{34}	2^{31}	$2^{36.6}$	0.99
MitM [23]	AES-128	5	2^{3}	2^{64}	2^{56}	-
Integral [24]	AES-128	5	2^{40}	$2^{38.7}$	2^{40}	–
MDC [12]	AES-128(Small Variant)	5	$2^{33.6}$	$2^{33.28}$	2^{34}	–
MDC [25]	AES-128	5	$2^{24.6}$	$2^{25.5}$	2^{28}	0.65
Yoyo [26]	AES-128	5	$2^{11.5}ACP$	2^{31}	Small	–
Boomerang [27]	AES-128	5	2^{9}	$2^{16.5}$	2^{9}	≥ 0.50
Square [21]	AES-128	6	2^{35}	2^{72}	2^{32}	–
Truncated Boomerang [28]	AES-128	6	$2^{59}ACC$	2^{61}	–	–
Boomerang [11]	AES-128	6	2^{71}	2^{71}	2^{33}	–
Boomeyong [22]	AES-128	6	$2^{79.72}ACC$	$2^{78}XOR$	2^{28}	–
IDC [18]	AES-128	6	$2^{91.5}$	2^{122}	–	–
IDC [19]	AES-128	6	$2^{114.5}$	2^{46}	–	–
Collision [8]	AES-128	7	2^{32}	2^{128}	–	–
Collision [8]	AES-192/256	7	2^{32}	2^{140}	–	–
Zero-diff [29]	AES-128	7	$2^{110.2}$	$2^{110.2}$	$2^{110.2}$	–
IDC [30]	AES-128	7	2^{106}	2^{110}	$2^{90.2}$	–

Table 1 (continued)

Attack	AES version	Rounds	Data complexity	Time complexity	Memory	Success Prob.*
IDC [20]	AES-192	7	2^{92}	2^{186}	2^{153}	–
IDC [20]	AES-256	7	$2^{92.5}$	2^{250}	2^{153}	–

*Here, – means the data is not given in the corresponding source

1.1.5 Yoyo attack

Biham et al. [13] first introduce the concept of yoyo attack for the cryptanalysis of the cipher Skipjack. This attack first finds a zero/nonzero difference property in a cipher path with a plaintext pair. Next, swapping the portions of the corresponding ciphertext pair generates a new ciphertext pair, which creates a new plaintext pair after decryption. The new plaintext pair follows the original zero/nonzero difference property. Assume a plaintext pair (P_1, P_1') each of size 16 bits, with some fixed difference. After encryption of some rounds let the ciphertext be (C_1, C_1'). Now, we swap the first four bits of C_1 and C_1' to get a new ciphertext pair (C_2, C_2'), which we decrypt to get plaintexts (P_2, P_2'). Next, swap the second four bits of the two plaintexts to get new plaintext pair (P_3, P_3'), and after encryption, the ciphertext pair is (C_3, C_3'). Repeat this process multiple times to get new such pairs. Here, all of the (P_1, P_1'), (P_2, P_2') and (P_3, P_3') maintains same property, thus generating many plaintext/ciphertext pairs called adaptively chosen plaintext/ciphertext pairs. The cryptology researchers are now using property to attack block ciphers like AES (see Table 1).

1.1.6 Integral attack

The concept of integral cryptanalysis [14] proposed by Knudsen and Wagner, in FSE 2002. The main purpose is to find the propagation of sums (values) in the cipher path. The attacker chooses d bit positions from the plaintexts and sets other positions as any constants. Next, construct a set of 2^d plaintexts (called d^{th} order integral). Now check how a fixed sum of the generated set progresses through the cipher path after applying the encryption algorithm. The main idea is to find the number of rounds this property exists and accordingly mount the attack. In 2020, Grassi et al. [15] proposed a combination of mixture differential attack and zero-sum distinguisher, called mixture integral attack.

1.1.7 Meet in the middle attack

Diffie and Hellman [16] first proposed the concept of Meet in the Middle Attack (MITM). Next Dimirci et al. [17] applied on reduced rounds of AES. In MITM, initially the cipher is divided into three parts, $\mathrm{E} = E_1^{K_1} \circ E_2 \circ E_3^{K_3}$. Here the adversary knows an integral (sum) property for the encryption E_2 corresponding to a set, called δ-set. Then he/she finds all possible values using E_2 by applying the δ -set and forms a precomputed table T. After choosing the appropriate plaintext-ciphertext pairs, the attacker guesses K_1 and performs encryption using E_1 to get the input δ-set. Next, guess K_2 and execute partial decryption of the corresponding ciphertexts using E_3. Now check if the values occur in table T. If not, discard the key pairs (K_1, K_2), thus reducing the original key-space.

Table 2 Time required for key recovery of AES

AES Version	Round	Attack	System used	Time taken	Recovery type
AES-128	5	Kakarla et al. [31]	Centralized	48 h	Full Key
		Section 3	Distributed	6.5 min	
AES-192/256	5	Kakarla et al. [31]	Centralized	50 Hours	
		Section 4	Distributed	10 min	
AES-192/256	6	Section 5	Distributed	12.5 min	Partial Key

1.2 Study of related work

Biham [7] and Knudsen [2] first propose the impossible differential approach for the cryptanalysis of Skipjack and DEAL, respectively. In the first attack using IDC on five-round AES-128, introduced by Biham and Keller [7], a total of $2^{29.5}$ chosen plaintexts are needed, the running time is 2^{31} encryptions, the memory requirement is 4 TB, and 2^{37} one-round encryptions are used as precomputation. Cheon et al. [18] improve the attack after applying it to six-round AES-128. This attack requires 2^{122} encryptions and $2^{91.5}$ chosen plaintexts. Zhang et al. [19] further reduce the number of encryptions to 2^{46}, but the number of chosen plaintexts rises to $2^{114.5}$. An extended attack on seven-round AES-192 and AES-256 is given by Phan [20] in 2004. The attack for AES-256 uses $2^{250.5}$ seven-round encryptions, 2^{153} bytes of memory, and $2^{92.5}$ plaintexts. The attack on AES-192 needs 2^{186} seven-round encryptions, 2^{153} bytes of memory, and 2^{92} chosen plaintexts.

In Table 1, we summarize the important key-recovery attacks on AES-128, AES-192, and AES-256. Here, IDC and MDC represent impossible-differential and mixture-differential cryptanalysis, respectively.

1.3 Our contribution

We mainly investigate the memory efficient and parallel implementation process of impossible differential attacks, that is applied AES. Our purpose is to recover the partial/full key for five/six rounds of AES. Here, we report a cluster-based practical attack of five-round AES-(128, 192, 256)[1] and six-round AES-(192, 256) with high success rate. We use memory-optimization techniques needed for the precomputed data to work in our parallel environment. We distribute the whole work into multiple worker nodes, which are connected in a loosely coupled manner. The total memory requirement per node is reduced from 128.5 [31] to 96.5 GB in the case of AES-128. For effectively accessing the large tables, we use efficient data structures. To retrieve the full key, our attack takes 6.5 minutes for AES-128 and 10 minutes for AES-256 and AES-192. For second-round key recovery of six-round AES-(192, 256), it takes 12.5 minutes. Tables 1 and 2 summarize briefly our contribution.

[1] A preliminary version of this paper is presented in SPACE 2019 [32].

Fig. 1 4 X 4 Byte positions of an 128 bit AES state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1.4 Organization of the paper

We organize the paper as follows. Section 2 describes the required background, a short description of the AES algorithm, all the notations used in the whole paper, the basic four-round impossible-differential characteristics of AES, and the original impossible differential attack of Biham [7] on round-reduced AES. Section 3 provides our practical attack using IDC on five-round AES-128. Section 3.1 explains the data- and task-parallelism concepts elaborated in our attack. Section 3.3 deals with the step-by-step algorithm of the attack procedure. Our experimental results are presented in Sect. 3.4. In Sect. 4, we extend the attack to work for five-round AES-192 and AES-256. Section 5 explains briefly our extended attack on six-round AES-192 and AES-256. In Sect. 6, we briefly describe the significance and advantages of our work. Section 7 concludes the paper.

2 Impossible differential cryptanalysis of AES

2.1 A short description of the AES

AES is a symmetric-key block cipher with 128-bit blocks and uses three variants of key sizes. AES-128 uses 128-bit keys and 10 rounds, AES-192 uses 192-bit keys and 12 rounds, and AES-256 uses 256-bit keys and 14 rounds. The input or output of any round is a 128-bit block which is represented as a 4×4 byte state matrix as given in Fig. 1. A round in AES consists of four operations.

- *SubBytes (SB)* Sub-bytes transformation is a nonlinear operation in AES which works in byte level of the state matrix.
- *ShiftRows (SR)* This shifts left each byte by one cell in the first row, by two cells in the second row, and three cells for the third row. No shift is performed on the bytes in the zeroth row.
- *MixColumns (MC)* It is a linear transformation which operates on each column of the state matrix. Here, a 4×4 MixColumns matrix is multiplied to each column.

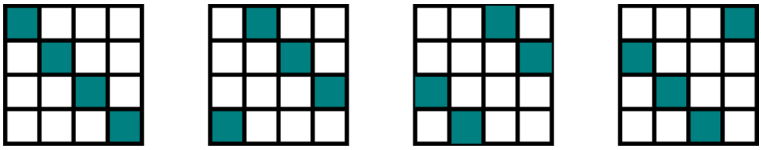

Fig. 2 Plaintext diagonal positions

- *AddRoundKey (AK)* This involves bit-by-bit XORing of the 128-bit the round key with the state matrix.

The last round omits the MixColumns operation. Also, AES encryption uses an initial key XORing operation before starting the first round. The decryption operation is just the reverse of the encryption process. AES uses a key-expansion algorithm to generate the round keys needed in the AddRoundKey operations. Each round key is a 128-bit block in all the versions of AES.

2.2 The notations used

The notations used in our attack we define as follows

- P: Plaintext
- C: Ciphertext
- S_i^{SR}: Denotes the temporary matrix after applying ShiftRows operation of the i-th round.
- S_i^{SB}: Denotes the temporary matrix after the SubBytes operation of the i-th round.
- S_i^{MC}: Represents the temporary matrix after the MixColumns operation of the i-th round.
- S_i^{AK}: Denotes the temporary matrix after the AddRoundKey operation of i-th round (we have $S_i^{AK} = S_{i+1}$).
- S^b: The b-th byte in any state matrix for $b \in \{1, 2, \ldots, 16\}$ (see Fig. 1).
- K_0: The initial key.
- K_i: Denotes i-th round key($i \geq 1$).
- P^D: Represents the set of plaintext diagonals (shown in Fig. 2 by filled boxes) which have the four byte combinations as follows:

$$P^D = \Big\{(1, 6, 11, 16), (2, 7, 12, 13), (3, 8, 9, 14), (4, 5, 10, 15)\Big\}.$$

- C^D: The set of ciphertext diagonals (given in Fig. 3 by unfilled or white boxes) which have the four byte combinations as follows:

$$C^D = \Big\{(1, 8, 11, 14), (2, 5, 12, 15), (3, 6, 9, 16), (4, 7, 10, 13)\Big\}.$$

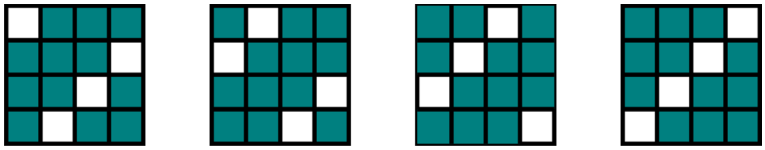

Fig. 3 Ciphertext diagonal positions at the end of fifth round

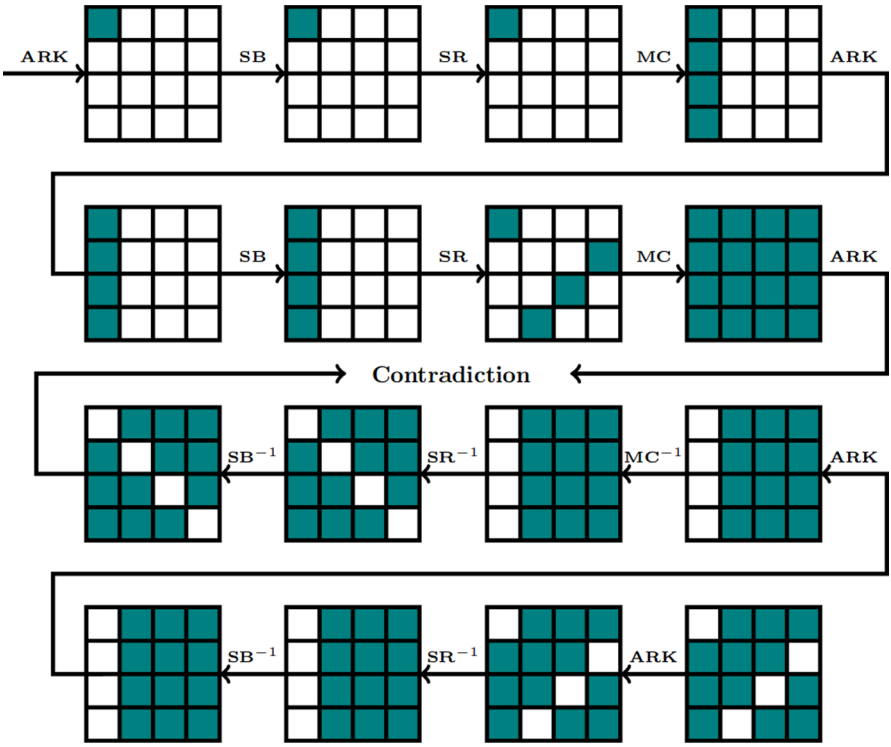

Fig. 4 IDC Path of four-round AES

2.3 Four-round impossible-differential cryptanalysis of AES-128

This section briefly explains the four-round impossible-differential property is given by Biham and Keller [7] to describe the five-round attack on AES-128. Most of the related research uses this property to mount higher-round attacks using IDC. We denote a nonzero byte as an *active byte* after XORing any two-state matrices. If the XOR results in a zero byte, we call that byte a *passive byte*. In the impossible-differential path of four-round AES, the active and the passive bytes are shown in Fig. 4. The filled boxes represent the active bytes, and the white boxes represent the passive bytes.

Property 1 (MixColumn property) An AES state, having only one byte of nonzero difference in a particular column, after applying MixColumns the corresponding column contains all nonzero differences.

Let P_1 and P_2 be two chosen plaintexts which provide one active byte as shown in Fig. 4. By the MixColumn property, all bytes of the state matrix become active after two rounds of encryption. If C_1 and C_2 are two chosen ciphertexts which have passive bytes from the set C^D as in Fig. 3, then we get the corresponding state matrix with passive bytes in the diagonal positions in P^D (with interchanged active and passive bytes as in Fig. 2). With this observation, we state the following property for future use.

Property 2 (Impossible-differential property) When two plaintexts with equal in all bytes except one, then after application of four rounds of AES encryption, the two resulting ciphertexts cannot contain zero differences in any one of the diagonals of C^D.

2.4 Impossible-differential attack on five-round AES-128

In this section, we mainly focus the IDC of Biham and Keller [7] applied to five-round AES-128. They use the four-round impossible-differential path (see Fig. 4) to mount the five-round IDC. Now, at the top of the four-round impossible-differential path one more round is included as shown in Fig. 5.

Chosen pairs are the plaintext pairs that provide active bytes (after XORing) in any one of the diagonals from the set P^D. After five rounds of encryption of the chosen pairs, if we get ciphertext pairs with zero differences in one diagonal which belongs to one of the diagonal positions from the set C^D, we call those chosen pairs as *desired pairs*.

2.5 Foundation of our attack

Let P_1, P_2 be a desired pair, and C_1, C_2 the corresponding ciphertexts generated after five-round encryptions, obtained by the encryption oracle. $\Delta C = C_1 \oplus C_2$ produces a state matrix with passive bytes in one of the ciphertext diagonals from the set C^D. We also denote $\Delta P = P_1 \oplus P_2$. Let Q_1, Q_2 be the two state matrices after the application of the MixColumns operation of the first round. $\Delta Q = Q_1 \oplus Q_2$ has only one active byte. Now, we apply the one-round decryption operation (using the MC^{-1}, SR^{-1}, and SB^{-1} operations only) with input Q_1, Q_2 (let the output be R_1, R_2). Here, the ARK operation is not involved. We can check that the resulting XOR difference $\Delta R = R_1 \oplus R_2$ must have one active diagonal from the set P^D as given in Fig. 6.

In the desired pair P_1, P_2, no key XORing is done. But R_1, R_2 is the result after the initial key XORing with K_0. Therefore $P_1 \oplus R_1$ and $P_2 \oplus R_1$ generate two impossible 32-bit values of the initial key K_0 for the byte positions as in the set P^D. These 32-bit impossible key values are called *invalid partial keys*. According to four non-overlapping plaintext diagonals (from the set P^D), we calculate four partial-key sets. After merging those invalid partial-key sets, we get the full invalid keys; thereby, the keyspace is reduced.

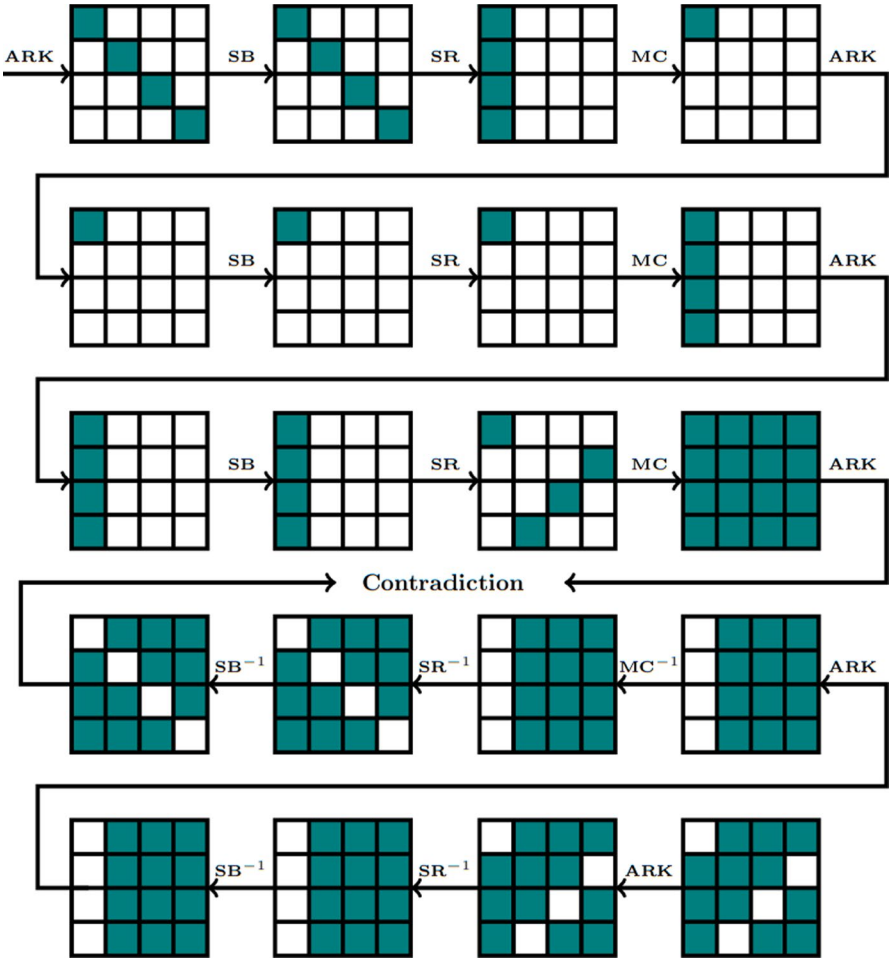

Fig. 5 Five-round Impossible-differential Path

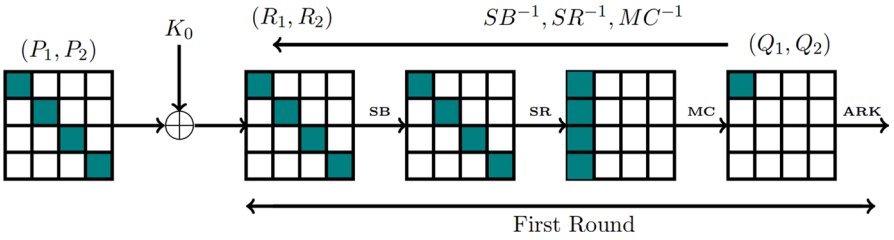

Fig. 6 Invalid partial key elimination at first round

Form one column of Q_1 and Q_2 as (w, x, y, z) and (w', x, y, z) with $w \neq w'$. This is the only column of Q_1 and Q_2 that contains the active byte. The total number of

choices of Q_1, Q_2 is about 2^{40}, whereas ΔR has 2^{32} possibilities. Therefore for each value of ΔR, there are 2^8 choices of Q_1, Q_2 pairs. With only one active diagonal, there are 2^{32} possible plaintexts. The number of chosen pairs with such plaintexts is $\binom{2^{32}}{2} \approx 2^{63}$. For each such pair, the probability of getting one ciphertext with passive diagonal positions from the set C^D becomes 2^{-32} after five rounds encryption of AES. But due to four ciphertext diagonals, the new probability becomes 2^{-30}. Thus the expected number of desired pairs is $2^{63} \times 2^{-30} = 2^{33}$. Here we discover two invalid partial keys with probability 2^{-32}. Hence, after this filtering with 2^{32} chosen plaintexts, the expected number of invalid partial keys which are not discarded is about

$$2^{32}\left(1 - 2^{-32}\right)^{2^{33} \times 2^8} \approx 0 \tag{1}$$

According to Biham and Keller [7], successful execution of this attack requires $2^{29.5}$ chosen plaintexts and thus 2^{28} desired pairs. The precomputation cost is about 2^{37} one-round decryptions. For key elimination, 2^{31} five-round encryptions are required. Also, about 4 TB memory is needed.

For this attack, two tables are generated in the precomputation phase. In the first table, we store all the desired pairs in the following way. We fix a plaintext diagonal from the set P^D and a ciphertext diagonal from the set C^D. Let P be a plaintext state matrix with active bytes in one of the plaintext diagonals, and p the 32-bit value corresponding to the diagonal. Using the five-round encryption oracle on P, we get the ciphertext C with the 32-bit ciphertext diagonal position value as c.

In the *desired-pair table*, we store p indexed by c. In case of multiple instances p_1, p_2 stored in the same index, c corresponding to a plaintext pair (P_1, P_2) for which $\Delta C = C_1 \oplus C_2$ contains passive bytes in the given ciphertext diagonal set. The desired-pair table eliminates the wrong partial keys by forming a second table.

The second table, denoted as *hash table*, is generated as follows. We consider all possible 2^{40} combinations of Q_1, Q_2, find the corresponding R_1, R_2, and compute $\Delta R = R_1 \oplus R_2$. From the diagonal positions of R_1 and ΔR, the four byte values taken out is denoted as r_1 and δ_r respectively. We store r_1 corresponding to index δ_r. Hence, for each hash-table entry the average size of chained list is 2^8.

3 Attack on five-round AES-128 for full key recovery

We make use of a clustered system which consists of eight worker nodes configured with one master node to execute the IDC algorithm. The nodes do not share any memory, so the required data must be copied to each node. Each worker node has 32 processing cores which shares the data stored in the common memory of the particular node.

The top-level view of our five-round attack is given in Fig. 7. The hash table is calculated in the precomputation phase by one-round decryption starting at S_1^{MC}. Also, the desired-pair table is generated by running five-round encryption oracle. All the precomputations are done parallelly in the worker nodes. The key-elimination step invalidates all the partial keys for each of the plaintext diagonal. This process

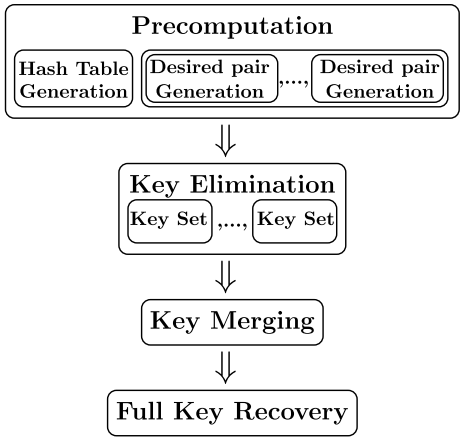

Fig. 7 Steps of the attack procedure

needs the hash table and the desired pair table and runs parallel at the eight worker nodes. Each worker node returns one reduced valid partial-key set. Thus, we have eight partial-key sets which are merged into four sets corresponding to four plaintext diagonal. Finally, in the full-key recovery phase, the four valid partial-key sets are distributed among the eight working nodes, which eventually merge the partial key sets to form a possible full valid key set. The correct full key is finally identified by doing a brute-force search on all the remaining possible candidates.

3.1 Parallel precomputation of data sets across the worker nodes

All the phases of precomputation and key elimination run in parallel in the eight worker nodes. These computations go on in a data-parallel fashion, as different nodes take non-overlapping data sets for computing the same task in parallel. In each worker node, the assigned task is subdivided into 32 sub-tasks and each sub-task is mapped to a single core.

3.1.1 Desired-pair table construction

We choose a particular plaintext diagonal to generate the desired-pair table and form an AES state P. By using the encryption oracle, we encrypt all possible 2^{32} states for five rounds. Let C be the ciphertext for P. We extract the 32-bit value of the plaintext P and the ciphertext C corresponding to the given plaintext diagonal and ciphertext diagonal. Let these 32-bit values by p and c. In the desired pair table, we store p at index c. The procedure is given in Algorithm 1. Step 1 initializes *DesiredPairTable* to empty. Step 2 runs a loop to all possible 32-bit values of (w, x, y, z) which requires 2^{32} number of iterations. Step 3 initialize an AES state with (w, x, y, z) and a given plaintext diagonal *PTDiag* and stores the plaintext to *PT*. In step 4 an encryption of five round AES occurs and assume the ciphertext as *CT*. Step 5-6 extracts the 32-bit plaintext and the corresponding ciphertext. The main cost for the algorithm is 2^{32} number of AES five round encryptions. For construction in parallel with eight

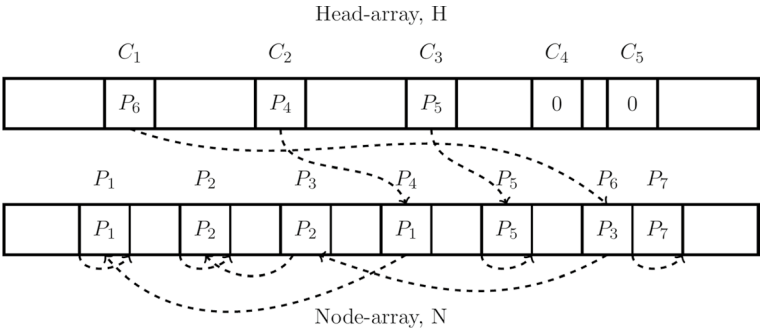

Fig. 8 Chained plaintexts indexed by ciphertexts during desired-pair table construction

worker nodes, we break the set of 2^{32} plaintext values to a subset of size 2^{29}, and distribute one subset to each node.

We use special data storage using linked lists for the construction of the desired-pair table. All the plaintexts p which generate the same ciphertext c are stored in the form of a linked list in an array called the *child array* (see Fig. 8). The starting index of such a list is stored in an array at the index specified by the generated ciphertext. This array is called the *head array*. For example, in Fig. 8, the ciphertext c_1 originates from the plaintexts p_2, p_3, p_6. Similarly, p_1, p_4 have the same ciphertext c_2.

Algorithm 1 Generation of the desired-pair table

Inputs: $PTDiag, CTDiag$
Output: $DesiredPairTable$

1: $DesiredPairTable \leftarrow Empty$
2: **for** all $(w, x, y, z) \in 0,1,\ldots,2^8 - 1$ **do**
3: $\quad PT \leftarrow InitializeState(w, x, y, z, PTDiag)$
4: $\quad CT \leftarrow$EncryptionOracle(PT)
5: $\quad CT \leftarrow Extract_{32}(w, x, y, z, CTDiag)$
6: $\quad PT \leftarrow Extract_{32}(w, x, y, z, PTDiag)$
7: $\quad$Append($DesiredPairTable[CT]$, PT)
8: **end for**
9: **return DesiredPairTable**

3.1.2 Hash-table generation

The hash table is formed using the following manner. We take all possible 2^{40} values of (w, w', x, y, z). By making one-round decryption, we calculate the states R_1, R_2. In the hash-table index $\delta_r = \text{Extract}_{32}s(\Delta R) = \text{Extract}_{32}(R_1 \oplus R_2)$, we place $r_1 = \text{Extract}_{32}(R_1)$. Here, Extract_{32} extracts 32-bit values from an AES state. We store the hash table in the same way as the desired-pair table, that is, using a head array and a child array. Now we describe Algorithm 2. Step 1 initializes hash-table

as empty. Step 2 traverses 2^{40} values of (w, w', x, y, z). Step 3-4 initializes two AES states Q_1 and Q_2. Step 5 and Step 6 both run one round of AES decryption, which dominates the main computation under the loop and stores the result in R_1 and R_2. In Step 7-8 we extract 32-bits of R_1 to r_1 and also 32-bits of $R_1 \oplus R_2$ to δ_r. Step 9 appends r_1 into the hash-table indexed by Δ_r. Here, the total cost is 2^{40} of one round AES encryption/decryption. We break the set of 2^{40} values to subsets of size 2^{37} and distribute one subset to each node for the eight worker nodes.

Algorithm 2 Hash table generation

1: $HashTable \leftarrow Empty$
2: **for** tuples (w, w', x, y, z) assigned to the worker node, **do**
3: $\quad Q_1 \leftarrow$InitializeState(w, x, y, z)
4: $\quad Q_2 \leftarrow$InitializeState(w', x, y, z)
5: $\quad R_1 \leftarrow SB^{-1}(SR^{-1}(MC^{-1}(Q_1)))$
6: $\quad R_2 \leftarrow SB^{-1}(SR^{-1}(MC^{-1}(Q_2)))$
7: $\quad r_1 = \text{Extract}_{32}(R_1)$
8: $\quad \delta_r = \text{Extract}_{32}(R_1 \oplus R_2)$
9: $\quad \text{Append}(Hashtable[\delta_r], r_1)$
10: **end for**
11: **return HashTable**

3.1.3 Hash-table division

For effective utilization of the node memories, the hash table is broken into two parts. There are 2^{32} rows in the hash table indexed by δr. We divide the hash table into two parts based on the MSB of δr. Let the two parts be H_0 and H_1. During key elimination, the eight worker nodes generate eight valid and reduced partial-key sets, each corresponding to one plaintext diagonal and one hash table part (H_0 or H_1). The exact mapping of the plaintext diagonals and the hash-table parts to the nodes is given in Fig. 9.

3.1.4 Distribution of data into worker nodes

Our cluster consists of eight worker nodes. To make the attack efficient and utilize our cluster properly we distribute the desired-pair table and hash table into the worker nodes. We have four desired-pair tables and two hash tables. During attack each worker node requires a desired-pair table and one part of the hash table for key elimination. Now we replicate the first desired-pair table to a pair $(Node_1, Node_2)$, second desired-pair table to $(Node_3, Node_4)$ and so on as depicted in Fig. 9. Also the $HahsTable_1$ is copied to odd numbered nodes and $HashTable_2$ in even numbered nodes. By distributing the data in this way each pair of nodes can work parallelly over a desired-pair table and a different hash-table part. We take an intersection of the valid keys received from a pair.

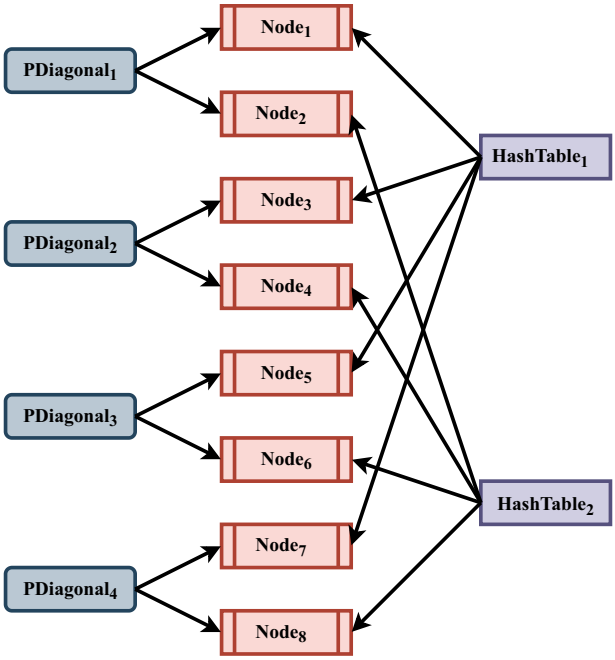

Fig. 9 Mapping of divided hash-table's and four desired-pair tables to the worker nodes

3.2 Optimization of memory requirement

The hash table maintains about 2^{40} four-byte values placed over 2^{32} indices, and each index holds 2^8 values on an average. The memory needed to store this hash table is $4 \times 2^{40} = 4$ TB, which is very high. Here, we propose some approaches by which the memory requirement can be reduced at the cost of an increased number of desired pairs. The running time continues to remain the same.

A simple observation reduces the hash-table size from 4 TB to 2 TB. Instead of taking both the pairs $((w, x, y, z), (w', x, y, z))$ and $((w', x, y, z), (w, x, y, z))$, we use only one pair by considering the condition $w > w'$.

Biham and Keller [7] mention that the attack can work even with $2^{29.5}$ chosen plaintexts (instead of 2^{32}), but the hash-table size remains at 2^{42} bytes. Kakarla et al. [31] introduces a data-memory trade-off. Let 2^m, $0 \leq m \leq 7$, be the average number of values per hash-table index. Therefore $2^m \times 2$ invalid keys are expected to be eliminated for each desired pair. If we consider 2^n possible desired pairs, the number of remaining partial keys can be calculated as

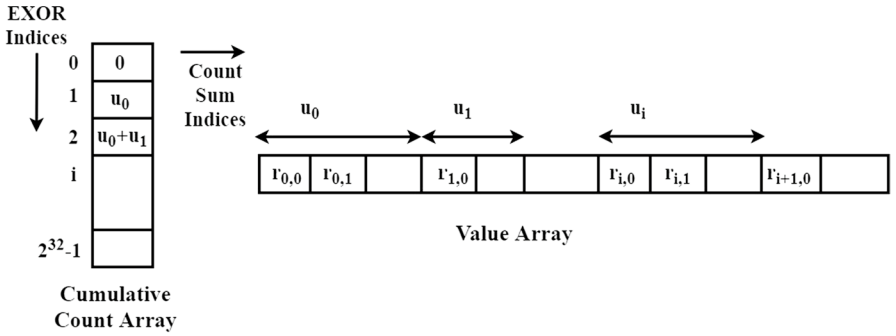

Fig. 10 Data structure for hash table accessing

$$2^{32}\left(1-2^{-32}\right)^{2^{m+1}\times 2^n}. \tag{2}$$

By the probability calculations of [7], we should have

$$m+n+1=36 \implies n=35-m. \tag{3}$$

This implies that the expected number of remaining partial keys is

$$2^{32}\left(1-2^{-32}\right)^{2^{m+1}\times 2^{35-m}} \tag{4}$$

Considering 2^{32} hash-table indices with each index storing an average of 2^m entries, the total number of hash-table values we calculate is $2^{32}\times 2^m$. To get 2^n desired pairs, we thus need $2^{(n+31)/2}$ chosen plaintexts. Moreover, the number of desired pairs needed is 2^{35-m} (because $n=35-m$). The number of chosen plaintexts required can be recalculated as $2^{33-0.5\times m}$. Taking $m=2$ leads to $2^{33-0.5\times 2}=2^{32}$ possible chosen plaintexts. Interestingly, the hash table now holds $2^{32+2}=2^{34}$ entries, and the memory size is reduced to $2^{34}\times 4$ bytes, that is, 64 GB.

3.3 Attack steps

In this subsection, we elaborate on the different steps of our attack.

3.3.1 Accessing precomputed data sets

- *Accessing the hash table* For accessing the hash table during key elimination, we maintain the following two arrays (see Fig. 10).

- *Value array* This array stores the r values. A total of 2^{32} r values are generated. For the choice of $m = 2$, there are about four entries for each index, so the value array stores a total of 2^{34} entries.
- *Cumulative-count array* Let the value array store u_r entries for a given value of r. For each value of r the cumulative-count array stores the cumulative partial sums; that is, the array stores the sequence $0, u_0, u_0 + u_1, u_0 + u_1 + u_2, \ldots, (u_0 + u_1 + u_2 + \ldots + u_{2^{32}-2})$.

During reading from the hash-table first search the cumulative count array to get the number of r values, which is u_r. Now to retrieve the actual r values we use the value array and fetch the u_r number of values.

- *Accessing the desired-pair table* The desired-pair table stores all the plaintexts indexed by the corresponding ciphertexts. For directly accessing the plaintexts during key elimination, we use the same approach as used in hash-table accesses. A value array stores the 32-bit plaintext values and cumulative-count array stores the cumulative counts of the plaintexts for different ciphertexts. The ciphertext works as the index for direct access.

3.3.2 Parallelism over plaintext diagonals

The key-elimination algorithm runs for four plaintext diagonals. All these diagonals are disjoint and can invalidate partial keys independently of one another. We distribute each plaintext diagonal to two worker nodes. More precisely, the i-th diagonal is replicated into the $(2i)$-th and the $(2i + 1)$-th worker node in the cluster.

3.3.3 Key elimination

We run this phase in eight worker nodes in parallel. Each worker node is supplied one plaintext diagonal from P^D, and one of the hash-table parts. For eliminating the invalid partial keys, we traverse the desired-pair table. For each desired pair (P_1, P_2), we calculate $\delta_r = \text{Extract}_{32}(P_1 \oplus P_2)$. We now search δ_r in the given hash-table part. If the search fails, another worker node handles this index. Otherwise, we fetch each r from the index δ_r of the hash table, and invalidate the partial keys $r \oplus \text{Extract}_{32}(P_1)$ and $r \oplus \text{Extract}_{32}(P_2)$. The key-elimination process for a worker node is described in Algorithm 3. In Step 1 we initialize the 32-bits key array. Step 2 selects any one of the ciphertext diagonals. In Step 2 we load the desired-pair table into DPT corresponding to a plaintext diagonal PT_{Diag} and ciphertext diagonal CT_{Diag}. Next we traverse the whole DPT at Step 4 and 5 to collect the plaintext pair (P_1, P_2). XORing P_1 and P_2 at Step 6 returns the hash-table index δ_r, which we use in Step 8 to fetch the hash-table value r. Next we take XOR r with 32-bit value of p_1 and P_2 separately to calculate the two different invalid keys at Step 9-10. Also we delete the two invalid keys from the valid key array. Step 16 returns the valid key array.

Algorithm 3 Key elimination at a worker node

Inputs: PT_{Diag}, $HashTable$
Output: $KeyArr$: Contains the valid keys

1: $KeyArr \leftarrow$AllKeys ▷ Initialize all possible four-byte keys as valid
2: **for** each ciphertext diagonal CT_{Diag}, **do**
3: $DPT \leftarrow LoadDesiredPairTable(PT_{Diag}, CT_{Diag})$
4: **for** $c = 0$ to $2^{32} - 1$ **do**
5: **for** $P_1, P_2 \in DPT[c]$ **do**
6: $\delta_r \leftarrow \text{Extract}_{32}(P_1 \oplus P_2)$
7: **if** δ_r belongs to $HashTable$ **then**
8: **for** $r \in HashTable[\delta_r]$ **do**
9: Invalidate $r \oplus \text{Extract}_{32}(P_1)$ from $KeyArr$
10: Invalidate $r \oplus \text{Extract}_{32}(P_2)$ from $KeyArr$
11: **end for**
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return**($KeyArr$)

3.3.4 Key-set merging and full-key recovery

We use the following steps for recovering the correct full key (see Algorithm 4).

- *Key-set merging* The output of the key-elimination stage is eight key sets obtained from eight worker nodes. In Step 2 of the algorithm, for each plaintext diagonal, we take the intersection of the two key sets obtained from the two nodes that handle that diagonal. These two nodes work on two different parts of the hash table independently. From the original key set, only those keys which are not eliminated by any of the two nodes are potential valid keys, and are kept for future use. We store the k^{th} diagonal partial key to $FPartialKey_k$.
- *Full-key recovery* After key-set merging, only 480–500 valid partial keys remain for each plaintext diagonal. At Step 4, we integrate the partial keys generated from four diagonals. Step 5 runs a brute-force search on the reduced valid key-set to recover the correct 128-bit key at Step 6. Step 7 returns the correct key.

Algorithm 4 Key-set merging and full-key recovery

Inputs: $PartialKey_{i,j} : i, j \in 0, 1, 2, 3;\ Ciphertext$
Output: key

1: **for** $k = 0$ to 3 **do**
2: $\quad FPartialKey_k = PartialKey_{k,0} \cap PartialKey_{k,1}$ ▷ Take intersection
3: **end for**
4: key=BuildFullKey($FPartialKey_0, FPartialKey_1, FPartialKey_2, FPartialKey_3$)
5: $out = AES128Encryption(plaintext, key, 5);$ ▷ Five-round encryption oracle
6: **if** $out = Ciphertext$ **then**
7: $\quad$ **return**(key)
8: **end if**
9: End

3.4 Implementation details and experimental results

We run the programs on a cluster with one master node and eight worker nodes, each with 32 cores running at 2.1 GHz. Worker nodes are equipped with 128 GB of RAM, and communicate with the master node using an InfiniBand switch. The used compiler is GCC (version 4.8.5), running with the POSIX-thread (pthread) API. It is also worth mentioning that all program compilation is performed using the -O2 optimization flag. We now work out the exact space usage and report the running times achieved by our attack.

- *Data complexity and space usage* The hash table requires 2^{32} entries and each entry contains on average four values, so the data complexity for the hash table becomes 2^{34}. For storing the full hash table, the total memory needed is $2^{34} \times 4 + 2^{32} \times 8$ bytes which is 96 GB. As the hash table is divided into two parts, and each plaintext diagonal is mapped to two nodes, each node can work on its part of the hash table. This reduces the size of the hash table to 48 GB per node for storing the value array and the cumulative-count array. For the desired-pair table we use 2^{32} chosen plaintexts corresponding to each plaintext diagonal. So, in this case the data complexity is 2^{34}. For accessing the desired-pair table, the value array requires $2^{32} \times 8$ bytes or 32 GB, and the cumulative-count array needs $2^{32} \times 4$ bytes or 16 GB. Thus the total space required for the desired-pair table is $32 + 16 = 48$ GB. Finally, the Boolean array *KeyArr* needs 0.5 GB, making a total memory requirement of about 96.5 GB per worker node. Table 3 summarizes these space-usage figures.

  Our strategy does not reduce the total space required for the implementation. Instead, having a loosely coupled system without any shared memory,

Table 3 Memory requirement at each worker node for five-round AES-128

Data structure	Memory requirement
Hash table	48 GB
Head array	16 GB
Plaintext array	32 GB
Key array	0.5 GB
Total	96.5 GB

we copy the hash-table across multiple nodes. However, we use the different desired-pair tables in different nodes.

- *Time complexity* The precomputation process of the hash table is key-independent and requires only 2^{36} one round decryption as a one-time cost. During desired-pair table formation, 2^{32} chosen plaintexts require for each plaintext diagonal. So, the overall data complexity is $4 \times 2^{32} = 2^{34}$ ciphertexts (oracle encryptions). For each desired pair from the desired-pair table in the key-elimination phase, we fetch four entries for each hash index of the hash-table on average. Hence, the expected number of XOR calculations require are $2^{33} \times 2^3 = 2^{36}$, which is approximately equals to the time of 2^{31} five-round encryptions [7].
- *Running time* Our key recovery process runs one thread from each node. For key elimination (Algorithm 3), each worker node takes two hours, and the precomputation time of the desired-pair table and hash-table is not included here. The brute-force search operation for finding the full key is allocated to each of the single threads from eight worker nodes. Considering this, the overall time required for finding the correct key by brute-force search is 10 minutes. By adding the times for key-elimination and brute-force search, the total time needed is, therefore, 2 hours 10 minutes. In case key-elimination and brute-force search can be done in a single node instead of eight worker nodes, then the total sequential time required will be 8×2 hours $+ 8 \times 10$ minutes ≈ 17.33 hours. This version of running time we have estimated manually and not implemented. For faster operations, we have carried out a multi-threaded implementation. Each node comprises 32 cores so which shares the computations for each node. Both of the algorithm key-elimination and key-merging, brute-force have parallelization potentials. At the time of running the key elimination phase, we face some synchronization and critical- section problems. When the threads access the *KeyArr* parallelly for invalidating the wrong subkeys during key elimination, then we should handle the concurrent write operations generated by multiple threads. By blocking the full *KeyArr* using a mutex (semaphore), all the writes proceed sequentially, which leads to a significant loss of parallelization. On the other side, blocking each element of *KeyArr* by separate mutex results in a huge number of mutexes. Hence, as a practical agreement, the KeyArr is sub-divided into

Table 4 Comparison of performance for full key recovery of five-round AES-128 using IDC

Attack	Data complexity	Memory used
Biham and Keller [7]	$4 \times 2^{29.5}$	4 TB
Kakarla et al. [31]	4×2^{32}	128.5 GB
This work	4×2^{32}	96.5 GB per node

Table 5 Comparison with respect to time required for full-key recovery

Attack	Type	System used	Time taken
Kakarla et al. [31]	Sequential	Centralized server	48 h
This work	Sequential	Single node, one thread	17 h 20 min
	Distributed	Eight worker and one master nodes, one thread per node	2 h 10 min
	Distributed	Eight worker and one master nodes, 32 threads per node	6.5 min

1024 equal-sized blocks. One mutex is used to guard one separate block, so we need a total of 1024 mutexes. Now, a thread locks the mutex corresponding to a block only when updating the status of a key in that block. This process enforces mutual exclusion during the concurrent write requests to the same block. As we are using 1024 mutexes and running 32 threads in parallel, so the collision probability is no more than 1/2 by the birthday paradox. Hence, the waiting period for a thread to acquire its lock is rather infrequent. Here, we effectively solved the concurrent-write problem. This version of attack results in a running time of 6 minutes for key-elimination and 30 seconds for brute-force search.

3.5 Performance comparison

Tables 4 and 5 present a comparative analysis of our attack with other existing results and implementations.

4 Full key recovery attack on five-round AES-192 and AES-256

In this section, we describe IDC on five-round AES-192 and five-round AES-256. The key-scheduling algorithm [21] of AES-192/256 implies that only the first-round partial keys cannot recover the full key. We also need the partial keys from the second round. Subsequently, by using the AES key-expansion algorithm (given as Algorithm 5), we can generate the partial keys of other rounds.

Algorithm 5 AES key-expansion

Inputs:

N_k = Length of key in 32-bit words; $N_k = 4, 6, 8$ for AES-128, AES-192, AES-256

Key= $Key[0], Key[1], \ldots, Key[N_k - 1]$ as the 32-bit words of the original key

N_b = Number of columns in state block

N_r = Number of rounds

Output:

W = Word array $W[0], W[1], \ldots, W[4N_r - 1]$ as the 32-bit words of the expanded key

1: **for** $i = 0$ to $N_k - 1$ **do**
2: $W[i] \leftarrow word(Key[4i], Key[4i + 1], Key[4i + 2], Key[4i + 3])$
3: **end for**
4: **for** $i = N_k$ to $N_b(N_r + 1) - 1$ **do**
5: $temp \leftarrow W[i - 1]$
6: **if** $i \mod N_k = 0$ **then**
7: $temp \leftarrow SubWord(RotWord(temp)) \oplus Rcon[i/N_k]$
8: **end if**
9: **if** $N_k > 6$ and $i \mod N_k = 4$ **then**
10: $temp \leftarrow SubWord(temp)$
11: **end if**
12: $W[i] \leftarrow W[i - N_k] \oplus temp$
13: **end for**
14: **return** W

4.1 Attack principle

The partial keys for the first-round key K_0 can be computed by applying the same approach as for five-round AES-128. We, therefore, assume that K_0 is available. In the rest of Sect. 4, we concentrate on how the second-round key K_1 can be recovered.

We use the differential path [31] shown in Fig. 11 to compute the partial keys for K_1. This path is a result of shifting forward the AES-128 five-round IDC path by one round (see Fig. 5).

Now we define the desired pairs differently. The ciphertext pairs which provide all active bytes except one are called the desired pairs. We also redefine the pattern for chosen plaintexts. The chosen plaintexts pairs for the five-round AES-128 path have one diagonal active from the set P^D. For any such diagonal, the application of one-round decryption using the first-round key K_0 makes all bytes active because of the first-round MixColumns operation. The plaintext pairs that provide one active diagonal from the set P^D after the first-round MixColumns (that is S_1^{MC}) are now taken as the chosen plaintexts.

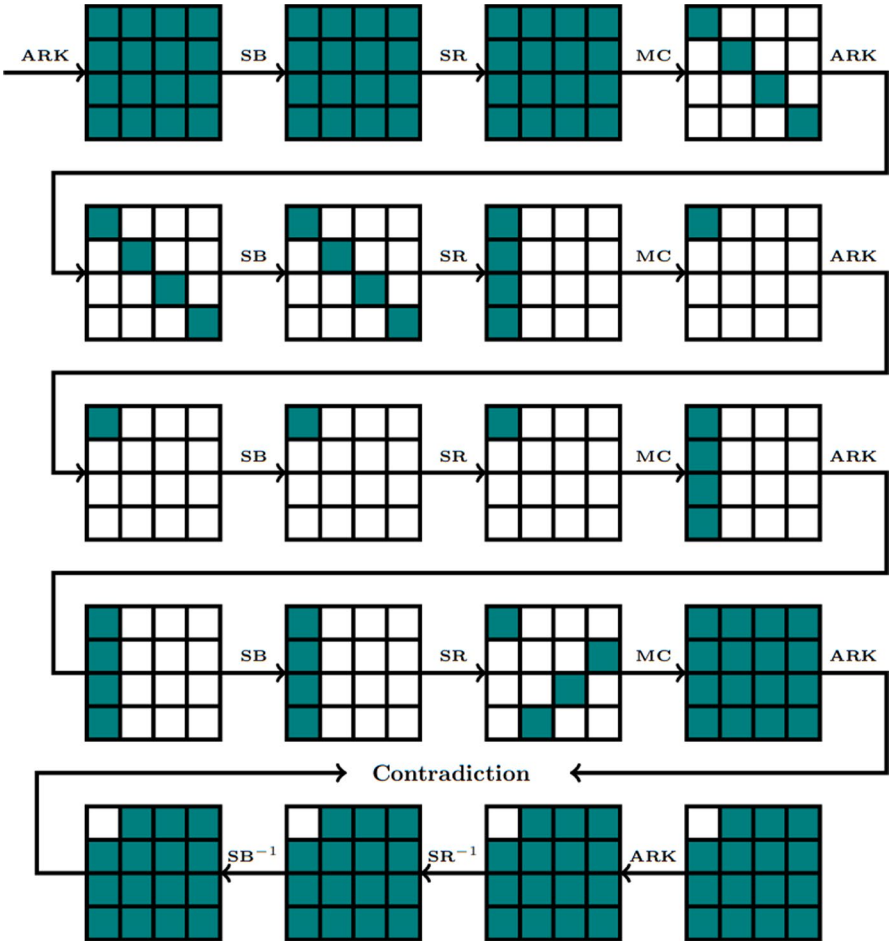

Fig. 11 Five-round path for generating the second-round partial key of AES-256/192

Let Q_1^1, Q_2^1 be the input state matrices of the second round such that $\Delta Q^1 = Q_1^1 \oplus Q_2^1$ has one active plaintext diagonal (see Fig. 12). Let δq^1 be the 32-bit value extracted from ΔQ^1. We take all possible 2^{32} values of δq^1, and perform one round of decryption by using the key K_0 to get the original plaintext pair P_1, P_2. Let C_1, C_2 be the ciphertext pairs after five-round encryption using the encryption oracle such that $\Delta C = C_1 \oplus C_2$ has only one active byte in the first row and the first column of the state matrix. Call the eight-bit extracted value from ΔC as δc. We store δq^1 indexed by δc in the desired-pair table.

We also need the hash table for key elimination. We take all possible values for Q_1^2, Q_2^2, and find the corresponding R_1^2, R_2^2 after one round of decryption (using the MC^{-1},SR^{-1},SB^{-1} operations). Let $\Delta R^2 = R_1^2 \oplus R_2^2$ have one active diagonal, and δr^2 the corresponding 32 bits extracted from ΔR^2. We store $r_1^2 = Extract_{32}(R_1^2)$ at the index δr^2 in the hash table. For multiple r_1^2 values mapped to the same index, we use

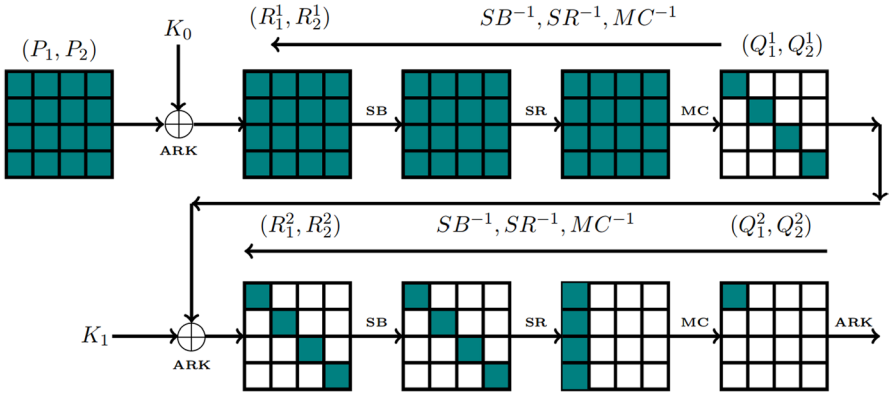

Fig. 12 Two-round path for generating the second-round partial key of AES-256/192

linked-list-based storage as explained earlier. Now, our goal is to find the second-round key K_1, provided that the first-round key K_0 is available. Here, we can check that $Q_1^1 \oplus R_1^2$ and $Q_2^1 \oplus R_2^2$ are two impossible partial keys for the second round. This attack technique works identically for AES-192 and AES-256.

4.2 Parallelism during precomputation

- *Parallelism during hash-table generation* We generate all of the 2^{40} values of (a, a', b, c, d) sequentially. Next, we find R_1^2, R_2^2 using one round of decryption, and append r_1^2 at index $\delta r^2 = Extract_{32}(R_1^2 \oplus R_2^2)$ of the hash table. Since we run an eight-node cluster, the set of 2^{40} values is divided into eight parts, each of size $2^{40}/8 = 2^{37}$, and each part is mapped to a node.
- *Parallelism during desired-pair generation* For each of the 2^{32} choices of Q_1^1, Q_2^1 with anyone active diagonal from P^D, we use one round of decryption K_0 to get the actual plaintext P_1, P_2. We invoke the encryption oracle to find the ciphertexts after five rounds of encryption. We choose only the ciphertexts C_1, C_2 with only one active byte and store δq^1 indexed by δc. The detailed algorithm for generating the desired pairs is given in Algorithm 6. In Step 2 we select all possible 32-bit plaintexts. Step 3 initializes the plaintext to a state variable Q. Assuming the first round key K_0 is already recovered we perform one round decryption of Q and store the temporary plaintext to P. Next execute a five-round AES random oracle on by taking input as P, which results in ciphertext C. In Step 6 we check if the first byte C is active and in case true we extract the active byte in CB at Step 7. Step 8 extracts the 32-bit diagonal of Q and stores it in the *DesiredPairTable* indexed by CB. The four plaintext diagonals can be handled independently of one another, so we can map the

diagonals to four worker nodes for generating the four desired-pair tables in parallel.

Algorithm 6 Generation of desired pairs for K_1 in five-round AES-192/256

Inputs:
$PTDiag$ stores the plaintext Diagonal
K_0 stores the first-round key
Outputs:
$DesiredPairTable$ stores plaintexts indexed by five-round encrypted ciphertext

1: $DesiredPairTable \leftarrow Empty$
2: **for** $PT = 0$ to $2^{32} - 1$ **do**
3: $Q \leftarrow Initializestate(PT, PTDiag)$
4: $P \leftarrow InvOneRound(Q, K_0)$ ▷ One-round decryption using K_0
5: $C \leftarrow EncryptionOracle(P)$ ▷ Five-round oracle encryption
6: **if** C has only the desired active byte **then**
7: $CB \leftarrow ExtractActiveByte(C)$
8: $DesiredPairTable[CB].Append(Extract_{32}(Q))$
9: **end if**
10: **end for**
11: **return** $DesiredPairTable$

4.3 The attack process

The issues relevant to the recovery of K_1 by IDC are now elaborated.

4.3.1 Reduction of data complexity

In the derivation of Sect. 3.3, the required number of hash-table entries is 2^{34}. Now, the desired-pair table has 2^8 indices and each index can hold a maximum of 2^{32} plaintexts. The total number of plaintexts can be 2^{40}, but we do not require many plaintexts. To cover all of the 2^{34} entries of the hash table, it suffices to take $\binom{2^{17}}{2}$ entries for each ciphertext index. So the total number of chosen plaintexts needed can be $2^{17} \times 256 = 2^{25}$. For four plaintext diagonals, the number of plaintexts needed is $4 \times 2^{25} = 2^{27}$.

4.3.2 Reduction of hash-table size

In Sect. 3.3, we calculated that the required number of hash-table entries is 2^{34}. In the desired-pair table, we choose the plaintexts in the range $[0, 2^{25} - 1]$ to generate the ciphertexts after five-round encryption by the oracle. At the time of key

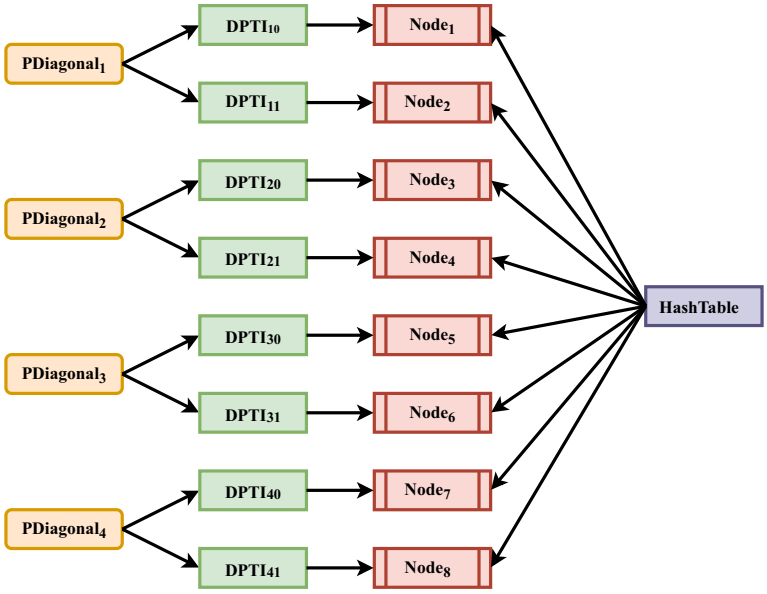

Fig. 13 Distribution of the hash table and parts of the desired-pair table to eight worker nodes for AES-192/256

elimination, the maximum value we need to search in the hash table is $2^{25} - 1$. It is therefore not needed to divide the hash table in two parts. Each node has sufficient memory to accommodate the entire hash table.

4.3.3 Division of the desired-pair table

Each index in a desired-pair table holds about 2^{17} entries. During key elimination, each desired pair can discard partial keys independently of one another. Each desired-pair table (corresponding to a plaintext diagonal) is broken down into two parts $DPTI_{i0}$ and $DPTI_{i1}$ for $i \in \{1, 2, 3, 4\}$. This division is for the msb of the indices. We thus get a total of eight parts of the four desired-pair tables to be distributed to the eight worker nodes (see Fig. 13). The output of the key-elimination stage is, therefore, eight valid partial-key sets.

4.3.4 Access of precomputed data

We continue to use the same organization of the hash table and the desired-pair table as illustrated in Fig. 10. The value array stores the actual entries, whereas the cumulative-count array quickly locates the first elements belonging to the indices.

4.3.5 Key elimination and recovery

- *Recovery of key sets* The key-elimination stage for the second-round key K_1 is performed at eight worker nodes in parallel. Each node gets half of the desired-pair table for a plaintext diagonal, and the entire hash table. The key-elimination procedure is given in Algorithm 7. In Step 1, we mark each bit of the valid key array *KeyValid* to true. Step 2 traverses the *DesiredPairsPart* and in each iteration collects two plaintext values $temp_1$ and $temp_2$. In Step 3, we calculate the *CountArr* index *pkey*, by XORing $temp_1$ and $temp_2$. Next, using *pkey* in Step 4 we read the range of the hash indexes. Step 5 to Step 8 fetches the hash value and finds the two invalid keys by XORing with plaintext values $temp_1$ and $temp_2$. Step 11 returns the valid key array *KeyValid*. The final output of this procedure is eight valid partial-key sets $PartialKey_{ij}$ for $i \in \{1, 2, 3, 4\}$ and $j \in \{0, 1\}$.
- *Merging of key sets* The master node collects the eight valid partial-key sets, and takes the intersection of $PartialKey_{i0}$ and $PartialKey_{i1}$ to generate four valid partial-key sets $FinalPartialKey_i$ corresponding to the four plaintext diagonals for $i \in \{1, 2, 3, 4\}$.
- *Retrieving the correct key* About 600 partial keys remain for each plaintext diagonal. We merge them and apply brute force on this reduced keyspace to obtain the full second-round key K_1.

Algorithm 7 Key elimination in a worker node for the second-round key K_1 of five-round AES-192/256

Inputs:
$DesiredPairsPart$ stores plaintexts indexed by five-round encrypted ciphertexts
$CountArr$ stores the cumulative counts for accessing hash-table entries
$HashArr$ stores the hash-table entries
Outputs:
The array $KeyValid$ to store the keys not discarded

1: Set each bit of $KeyValid$ to 1
2: **for each** $temp_1, temp_2$ in $DesiredPairsPart$, $temp_1 \neq temp_2$ **do**
3: $\quad pkey = temp_1 \oplus temp_2$
4: $\quad$ **for all** $i \in [CountArr[pkey], CountArr[pkey + 1] - 1]$ **do**
5: $\quad\quad value = HashArr[i] \oplus temp_1$
6: $\quad\quad$ Set $KeyValid[value] = 0$ ▷ Invalidate partial key
7: $\quad\quad value = HashArr[i] \oplus temp_2$
8: $\quad\quad$ Set $KeyValid[value] = 0$ ▷ Invalidate partial key
9: $\quad$ **end for**
10: **end for**
11: **return** $KeyValid$

Table 6 Maximum memory requirement at each worker node for AES-192/256

Data structure used		Memory requirement
Hash table	Value array	128 MB
	Cumulative count array	128 MB
Desired pair table		0.5 MB
Key array		512 MB
Total memory		768.5 MB

Table 7 Time and memory requirement for IDC on five-round AES-192/256

Recovered key	Chosen plaintexts	Memory	Time
K_0	$2^{32} \times 4 = 2^{34}$	96.50 GB	6.5 min
K_1	$2^{25} \times 4 = 2^{27}$	768.5 MB	3.5 min
K	$(2^{32} + 2^{25}) \times 4 \approx 2^{34}$	96.5 GB	10 min

4.4 Performance analysis and experimental results

We implement the IDC algorithm on the same cluster and in the same environment as described in Sect. 3.4.

- *Data complexity* According to Algorithm 2, the hash-table size is 2^{32} and the number of elements is 2^2. Hence total data complexity for the hash table precomputation is 2^{34}. In case of desired pair table generation, for the recovery of K_0 the required number of chosen plaintexts is the same as the five round AES-128, that is 2^{34}. But, for the recovery of K_1, we need to check the total number of hash table values which is 2^{34}. As the total number of entries in the desired-pair table is 2^8, we only choose 2^{17} elements from each ciphertext index. Hence, the total chosen plaintexts needed for four plaintext diagonals is $2^{17} \times 2^8 \times 2^2 = 2^{27}$. To sum up, for full key recovery the total chosen plaintexts required is 2^{34}.
- *Space usage* The desired-pair table is divided into two parts to utilize the eight worker nodes of the cluster fully. Thus, each partial diagonal gets two nodes to work on two different parts of the desired-pair table. This reduces the space of value arrays per node to 512 KB. The hash table (which is now reduced in size) requires 128 MB for each value array and cumulative count array. Thus a total of 256 MB requires for each node to load the hash table. Finally, the Boolean array storing the information about valid partial keys takes 0.5 GB. Therefore the total memory requirement is no more than 768.5 MB per node for finding second-round key K_1 (see Table 6).

  It is to be noted that the recovery of K_0 requires a much larger amount of space than the recovery of K_1. The recovery of K_1 starts after K_0 is available. As a result, the space for the recovery of K_0 (a small part of it actually) can be reused during the recovery of K_1.
- *Time complexity* For recovery of K_0 we need 2^{31} AES encryptions as described in Sect. 3.4. Next for recovery of K_1 we use Algorithm 7. Step 2 needs 2^{25} iterations

to fetch plaintexts from desired-pair table. In Step 4, for reading four values from each hash-table entry four iterations are needed. To sum up, the total number of XOR operations performed is $2^{25} \times 2^2 \times 2 = 2^{28}$.

- *Running Time* Table 7 reports the timing results of our implementation running on 32 available cores in each of the eight worker nodes. The time needed for finding the first-round key K_0 is 6.5 minutes as explained in Sect. 3.4. The key-elimination stage for the second-round key K_1 takes about 2 minutes of parallel running time in each node. The final brute-force If distributed to the eight worker nodes, the search takes a parallel running time of about 1.5 minutes. So the total time needed for retrieving the second-round key is about 3.5 minutes. Since the recovery of K_0 The recovery of K_1 proceeds sequentially; the total running time of our key recovery using IDC for AES-192/256 is $6.5 + 3.5 = 10$ minutes.

5 Partial-key recovery attack on six-round AES-192 and AES-256

In this section, we explain IDC briefly on six-round AES-192 and AES-256 for partial key recovering. From the AES key-scheduling algorithm [21], we know for AES-192/256, only the 128 bit first-round partial keys cannot recover the full key. We also need the 64 bit key of the second round for AES-192 and 128 bit second round key for AES-256. Subsequently, by using the AES key-expansion algorithm (given as Algorithm 5), we can generate the partial keys of other rounds.

5.1 Attack principle

Here, we assume that the first round key K_0 is available, and in the rest of this section, we concentrate on recovering the second round key K_1.

We use a new differential path shown in Fig. 14 to compute the partial keys for K_1. We create the new path by shifting forward the AES-128 five-round IDC path by one round (see Fig. 5), i.e., adding a new round at the front.

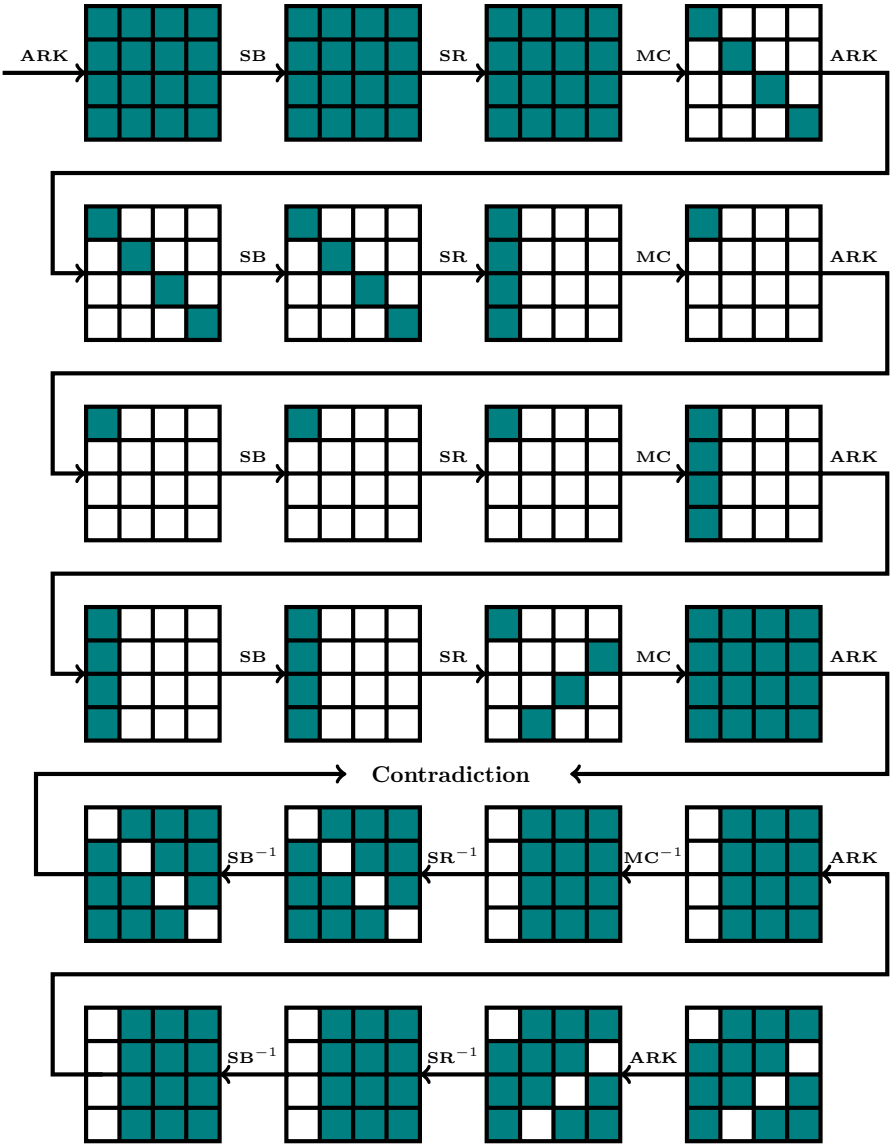

Fig. 14 Six-round path for generating the second-round partial key of AES-256/192

Table 8 Time and memory requirement for IDC on six-round AES-192/256

Recovered key	Chosen plaintexts	Memory	Time
K_1	$2^{32} \times 4 = 2^{34}$	96.50 MB	12.5 min

Algorithm 8 Generation of desired pairs for K_1 in six-round AES-192/256

Inputs:
$PTDiag$ stores the plaintext Diagonal
K_0 stores the first-round key
Outputs:
$DesiredPairTable$ stores plaintexts indexed by five-round encrypted ciphertext

1: $DesiredPairTable \leftarrow Empty$
2: **for** $P = 0$ to 2^{32} **do**
3: $Q \leftarrow Initializestate(P, PTDiag)$
4: $PT \leftarrow InvOneRound(Q, K_0)$ ▷ One-round decryption using K_0
5: $CT \leftarrow EncryptionOracle(PT)$ ▷ Six-round oracle encryption
6: $CT \leftarrow Extract_{32}(CT, CTDiag)$
7: $PT \leftarrow Extract_{32}(PT, PTDiag)$
8: $DesiredPairTable[CT].Append(PT)$
9: **end for**
10: **return** $DesiredPairTable$

The attack procedure is almost the same as the five-round AES-192/256, described in Sect. 4. The main difference is in generating the chosen-ciphertext pairs. Here, the chosen pairs belong to any one of the four diagonals from C^D. As given in Fig. 12, Q_1^1, Q_2^1 is the input state matrices of the second round and $\Delta Q^1 = Q_1^1 \oplus Q_2^1$ has one active plaintext diagonal. Let δq^1 be the 32-bit value extracted from ΔQ^1. We take all possible 2^{32} values of δq^1, and perform one round of decryption by using the key K_0 to get the original plaintext pair P_1, P_2. Let C_1, C_2 be the ciphertext pairs after six-round encryption using the encryption oracle such that $\Delta C = C_1 \oplus C_2$ has only one diagonal active bytes. Call the thirty two-bit extracted value from ΔC as δc. We store δq^1 indexed by δc in the desired-pair table. We need 2^{32} chosen plaintexts to get 2^{33} desired pairs to eliminate almost all of the wrong partial keys of K_1.

The algorithm for desired-pair generation is described in Algorithm 8. In Step 1, we initialize *DesiredPairTable* as empty. Next, we guess all possible 2^{32} plaintext values using the loop of Step 2 and corresponding to a plaintext diagonal *PTDiag* initialize an AES state Q in Step 3. Now, assuming the first round key K_0 is available Step 4 performs one round of decryption of Q and stores the result in *PT*. Step 5 call the AES encryption oracle of six-round and saves the ciphertext in *CT*. Next, Step 6 and Step 7 extracts the 32-bit plaintext and the corresponding 32-bit ciphertext diagonals to *PT* and *CT* accordingly. In Step 8, we store *PT* indexed by *CT* in *DesiredPairTable*.

The key-elimination and recovery of key works in a similar way as five round AES-192/256. This attack technique works identically for AES-192 and AES-256.

5.2 Performance analysis and experimental results

We implement the IDC algorithm on the same cluster and in the same environment as described in Sect. 3.4.

- *Space complexity* For accessing the desired-pair table, the value array requires $2^{32} \times 8$ bytes or 32 GB, and the cumulative-count array needs $2^{32} \times 4$ bytes or 16 GB. Thus the total space required for the desired-pair table is $32 + 16 = 48$ GB. The size of hash table is 48 GB per node. The Boolean array *KeyArr* needs 0.5 GB, making a total memory requirement of about 96.5 GB per worker node.

  Hence, the memory requirement for the hash table and desired pair table is the same as the five-round AES-128 attack as in Sect. 3.4. Individual partwise memory break up is mentioned in Table 3.
- *Time complexity* For hash table generation, the precomputation process requires only 2^{36} one round decryption as a one-time cost. During desired-pair table formation, 2^{32} chosen plaintexts require for each plaintext diagonal. So, the overall data complexity is $4 \times 2^{32} = 2^{34}$ ciphertexts (six round oracle encryptions and one round decryption). During key-elimination, for each desired pair from the desired-pair table, we fetch four entries for each hash index of the hash-table on average. Hence, the expected number of XOR calculations require are $2^{33} \times 2^3 = 2^{36}$. Table 8 reports the timing results of our implementation running on 32 available cores in each of the eight worker nodes. The key-elimination stage for the second-round key K_1 takes about 8 minutes of parallel running time in each node. The final brute-force search if distributed to the eight worker nodes takes a parallel running time of about 4.5 minutes. So the total time needed for retrieving the second-round key is about 12.5 minutes.

5.3 Communication complexity

After the completion of the precomputation stage, the hash table and the desired-pair tables should be sent to the worker nodes for the key-elimination stage to start. These communication times are not included in the running times mentioned above. Even with a 56 Gbps InfiniBand switch, we obtain a communication delay of about 10–12 minutes. Ironically, this delay is almost equal or larger than the best computation times achieved. However, this overhead can be completely eliminated by precomputing and storing the relevant data in the local hard disk of each worker node.

6 Discussion

For AES the theoretical attacks are available up to eight rounds with impractical data and time complexity. The practical attacks are available up to six rounds. The researchers are mainly focusing on attacks on reduced rounds. With the new cryptanalysis techniques of reduced rounds it opens different research directions. The researchers can propose more efficient techniques to mount attacks on higher rounds and as well as different versions of a cipher.

The concept of differential cryptanalysis is a basic approach for the cryptographic designers as well as attackers. IDC is a strong attack which automatically inherits the properties of differential cryptanalysis. Hence, IDC is applicable for most of the symmetric ciphers. Here our approach of attack and the implementation techniques can be applied to other ciphers easily. We propose a novel way to distribute the data and tasks to the worker nodes in a very efficient manner. Different memory reduction strategies and optimized data structures are adopted for this attack.. We use efficient data structures for accessing the precomputed data using the hash based mechanism. The researchers can adopt this methodology for implementing attacks on other SPN block ciphers.

7 Conclusion

In this paper, we present a parallel attack procedure applying Impossible Differential Cryptanalysis attacks on five-round AES. We have worked on all the variants of AES and achieved the partial and/or full key recovery for AES-128, 192, and 256. Different memory reduction strategies and optimized data structures are exploited for this effective and practical cluster-based attack. To make faster key elimination, the data tasks are divided into proper segments and distributed into the cluster's worker nodes. Our attack is the fastest one with higher success rate for partial and/or full key recovery of five-round AES using the impossible differential attack to the best of our knowledge.

During the key elimination of our attack the communication complexity is an issue, which adds some extra time with the total attack time and needs to identified correctly. Another part is the synchronization and critical section problem that occurred during the key elimination phase, when multiple threads simultaneously invalidates different keys in the same key array. The attacker have to manage this properly.

Extension of the same attack process on higher rounds of AES, results in an unmanageable increase of the data and time complexity. Although many theoretical estimations and improvements are available in the literature for six and seven rounds of AES, the practical attacks are not yet attempted and remain as challenging open problems to the cryptology community.

Nowadays, a wide variety of technologies use AES-256. Hence, the practical attacks in this version open new directions of research. First of all, with respect to brute force approach we reduced the overall key-space by half, which is huge.

Using the same approach for AES-192, the key-space is reduced to 64 bits. In future, if technology permits more faster and efficient systems, then this reduced key-space will help. Also, it is a challenge for the researchers to recover more bits for AES-192/256.

The running time can be reduced by adding more worker nodes, though the number of worker nodes should be chosen as a power of 2, so that the work can be evenly distributed among the worker nodes. Here we have to remember that the number of shares of the hash table should be the number of nodes divided by the number of partial diagonals (four).

Acknowledgements We thank the anonymous reviewers for their valuable comments and advice, which immensely enhanced the quality of this paper. The work of this paper is supported by the Ministry of Electronics and Information Technology, Government of India.

References

1. Biham E, Shamir A (1993) Differential cryptanalysis of the data encryption standard. Springer, Berlin, Heidelberg
2. Knudsen L (1998) Deal - a 128-bit block cipher. In: NIST AES Proposal
3. Biham E, Biryukov A, Shamir A (1999) Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In: Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, Proceeding, pp 12–23https://doi.org/10.1007/3-540-48910-X_2
4. Boura C, Naya-Plasencia M, Suder V (2014) Scrutinizing and improving impossible differential attacks: Applications to clefia, camellia, lblock and simon. In: Sarkar P, Iwata T (eds) Advances in Cryptology - ASIACRYPT 2014. Springer, Berlin, Heidelberg, pp 179–199
5. Boura C, Lallemand V, Naya-Plasencia M, Suder V (2018) Making the impossible possible. J Cryptol 31(1):101–133. https://doi.org/10.1007/s00145-016-9251-7
6. Yang Q, Hu L, Shi D, Todo Y, Sun S (2018) On the complexity of impossible differential cryptanalysis. Secur Commun Netw 2018:7393401–1739340111. https://doi.org/10.1155/2018/7393401
7. Biham E, Keller N (2000) Cryptanalysis of reduced variants of rijndael. In: 3rd AES Conference 230
8. Gilbert H, Minier M (2000) A Collision Attack on 7 Rounds of Rijndael. In: The Third Advanced Encryption Standard Candidate Conference, April 13-14, New York, New York, USA, pp 230–241
9. Daemen J, Knudsen LR, Rijmen, V (1997) The block cipher square. In: Biham E (ed.) Fast Software Encryption, 4th international workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings. Lecture Notes in Computer Science, vol 1267, pp 149–165. Springer. https://doi.org/10.1007/BFb0052343
10. Wagner DA (1999) The boomerang attack. In: Knudsen LR (ed) Fast software encryption, 6th international workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings. Lecture Notes in Computer Science, vol 1636, pp 156–170. Springer. https://doi.org/10.1007/3-540-48519-8_12
11. Biryukov A (2004). The Boomerang Attack on 5 and 6-Round Reduced AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol 3373, pp 11–15. Springer. https://doi.org/10.1007/11506447_2
12. Grassi L (2018) Mixture differential cryptanalysis: a new approach to distinguishers and attacks on round-reduced AES. IACR Trans Symmetric Cryptol 2018(2):133–160
13. Biham E, Biryukov A, Dunkelman O, Richardson E, Shamir A (1998) Initial observations on skipjack: cryptanalysis of skipjack-3xor. In: Tavares, SE, Meijer H (eds) Selected Areas in Cryptography '98, SAC'98, Kingston, Ontario, Canada, August 17-18, 1998, Proceedings. Lecture Notes in Computer Science, vol 1556, pp 362–376. Springer. https://doi.org/10.1007/3-540-48892-8_27

14. Knudsen LR, Wagner DA (2002) Integral cryptanalysis. In: Daemen J, Rijmen V (eds) Fast Software encryption, 9th international workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers. Lecture Notes in Computer Science, vol 2365, pp 112–127, Springer. https://doi.org/10.1007/3-540-45661-9_9
15. Grassi L, Schofnegger M (2020) Mixture integral attacks on reduced-round AES with a known/secret s-box. In: Bhargavan K, Oswald E, Prabhakaran M (eds) Progress in Cryptology - INDOCRYPT 2020 - 21st international conference on cryptology in India, Bangalore, India, December 13-16, 2020, Proceedings. Lecture Notes in Computer Science, vol 12578, pp 312–331. https://doi.org/10.1007/978-3-030-65277-7_14
16. Diffie W, Hellman ME (1977) Special feature exhaustive cryptanalysis of the NBS data encryption standard. Computer 10(6):74–84. https://doi.org/10.1109/C-M.1977.217750
17. Demirci H, Taskin I, Çoban M, Baysal A (2009) Improved meet-in-the-middle attacks on AES. In: Roy BK, Sendrier N (eds) Progress in Cryptology - INDOCRYPT 2009, 10th international conference on cryptology in India, New Delhi, India, December 13-16. Proceedings. Lecture Notes in Computer Science, vol 5922, pp 144–156. Springer. https://doi.org/10.1007/978-3-642-10628-6_10
18. Cheon JH, Kim M, Kim K, Lee J, Kang S (2001) Improved Impossible Differential Cryptanalysis of Rijndael and Crypton. In: Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, Proceedings, pp 39–49. https://doi.org/10.1007/3-540-45861-1_4
19. Zhang W, Wu W, Feng D (2007) New Results on Impossible Differential Cryptanalysis of Reduced AES. In: Information Security and Cryptology - ICISC 2007, 10th International Conference, Seoul, Korea, November 29-30, Proceedings, pp 239–250. https://doi.org/10.1007/978-3-540-76788-6_19
20. Phan RC (2004) Impossible differential cryptanalysis of 7-round advanced encryption standard (AES). Inf Process Lett 91(1):33–38
21. Daemen J, Rijmen V (2002) The design of Rijndael: AES - The advanced encryption standard. Information security and cryptography, Springer. https://doi.org/10.1007/978-3-662-04722-4
22. Rahman M, Saha D, Paul G (2021) Boomeyong: embedding yoyo within boomerang and its applications to key recovery attacks on AES and pholkos. IACR Trans Symmetric Cryptol 2021(3):137–169. https://doi.org/10.46586/tosc.v2021.i3.137-169
23. Derbez P (2013) Meet-in-the-middle attacks on AES. (attaques par rencontre par le milieu sur l'aes). PhD thesis, École Normale Supérieure, Paris, France. https://tel.archives-ouvertes.fr/tel-00918146
24. Tiessen T, Knudsen LR, Kölbl S, Lauridsen MM (2015) Security of the AES with a secret s-box. In: Leander, G. (ed.) Fast software encryption - 22nd international workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, revised selected papers. Lecture Notes in Computer Science, vol 9054, pp 175–189. Springer. https://doi.org/10.1007/978-3-662-48116-5_9
25. Bar-On A, Dunkelman O, Keller N, Ronen E, Shamir A (2018) Improved Key Recovery Attacks on Reduced-Round AES with Practical Data and Memory Complexities. In: Advances in cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II, pp 185–212. https://doi.org/10.1007/978-3-319-96881-0_7
26. Rønjom S, Bardeh NG, Helleseth T (2017) Yoyo tricks with AES. In: Takagi T, Peyrin T (eds) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, Proceedings, Part I. Lecture notes in computer science, vol 10624, pp 217–243, Springer. https://doi.org/10.1007/978-3-319-70694-8_8
27. Dunkelman O, Keller N, Ronen E, Shamir A (2020) The Retracing Boomerang Attack. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, Proceedings, Part I. Lecture notes in computer science, vol 12105, pp 280–309, Springer. https://doi.org/10.1007/978-3-030-45721-1_11
28. Bariant A, Leurent G (2022) Truncated boomerang attacks and application to aes-based ciphers. IACR Cryptol ePrint Arch, 701
29. Bardeh NG, Rijmen V (2022) New key-recovery attack on reduced-round AES. IACR Trans Symmetric Cryptol 2022(2):43–62. https://doi.org/10.46586/tosc.v2022.i2.43-62
30. Mala H, Dakhilalian M, Rijmen V, Modarres-Hashemi M (2010) Improved Impossible Differential Cryptanalysis of 7-Round AES-128. In: Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15. Proceedings, pp 282–291. https://doi.org/10.1007/978-3-642-17401-8_20

31. Kakarla S, Mandava S, Saha D, Roy Chowdhury D (2017) On the Practical Implementation of Impossible Differential Cryptanalysis on Reduced-Round AES. In: Applications and Techniques in Information Security - 8th International Conference, ATIS 2017, Auckland, New Zealand, July 6-7. Proceedings, pp 58–72. https://doi.org/10.1007/978-981-10-5421-1_6
32. Pal D, Agrawal D, Das A, Chowdhury DR (2019) An Efficient Parallel Implementation of Impossible-Differential Cryptanalysis for Five-Round AES-128. In: Security, Privacy, and Applied cryptography Engineering - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, Proceedings, pp 106–122. https://doi.org/10.1007/978-3-030-35869-3_9

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.